

Object-Oriented  
Model Representation  
for Control Systems Analysis

*Wolfgang Kreutzer*

Technical Report COSC 5/88

Paper submitted to *Automatica*

# *Object-Oriented Model Representation for Control Systems Analysis*

*Karl-Johan Astroem*

Dept. of Automatic Control - Univ. of Lund, Sweden

&

*Wolfgang Kreutzer*

Dept. of Computer Science - Univ. of Canterbury, New Zealand

## **Abstract:**

*The representation of systems is a key issue in system theory and computer-aided control engineering. This paper discusses different ways to represent systems and suggests an approach based on object-oriented programming which admits hierarchical descriptions of system structure and behaviour. A prototype implementation of these ideas, hosted by a so called 'flavour system' for object oriented programming is described together with some experiences in using it.*

## **1. Introduction**

The notion of **systems** is an essential element of control theory, and it is also a key issue in computer-aided control engineering (CACE). Systems can be represented in many different ways. There are graphical representations like block diagrams, signal flow diagrams and bond graphs. There are also mathematical representations like state space models and input-output relations which come in many different forms, matrix fractions, impulse responses, frequency responses. When working with control systems it is frequently helpful to use several different representations of a system.

Only fairly primitive ways of system representation are used in current CACSD systems. Typical examples are the *MathLab* derivatives where systems are described by matrices. A slightly more sophisticated representation is used in the simulation language *Simnon*. This representation recognizes that a system has properties such as inputs, outputs and states. *Simnon* also allows a system to be described as an interconnection of subsystems, with the restriction that only "flat" interconnections are permitted.

This paper presents a more flexible way to describe systems, which is based on the methodology of object-oriented programming. It is shown that a general structural description of hierarchically connected systems can be constructed from simple ingredients, by making a system an object with properties such as name, inputs, outputs and subsystems.

To obtain a useful tool it is also necessary to add behavioural descriptions. This is done by creating new objects describing behaviour. A system

can then inherit both structural and behavioural aspects. Behaviour can be characterised in many different ways. A state description is one of the simpler alternatives. This can be covered by introducing the object *StateBehaviour* with properties *states*, *stateTransitionMap* and *outputMap*. The behavioural descriptions should also allow a given system to be defined by models of different complexity. Apart from a detailed quantitative description it is also useful to be able to deal with qualitative descriptions of behaviour.

The paper describes a small prototype implementation of these ideas, which is written in Scheme and admits hierarchical system representation and symbolic manipulation of system descriptions. Our experiments with this prototype program indicate that this approach is a feasible means to implement powerful CACE systems.

The paper is organized as follows. Some system representations in current CACSD packages are described in section 2. Requirements on system representations are given in section 3. Sections 4 and 5 deal with representation of system structure and behaviour. The prototype implementation itself is described in section 6, together with an example of its application. Some conclusions are drawn in section 7, and the appendices contain a summary of the flavour system's functionality as well as the prototype's source code.

## 2. Examples

Some typical examples of system representations used in current CACSD packages are given in this chapter.

### Matrix languages

Linear time-invariant systems can adequately be described by using arrays. Such systems are therefore conveniently handled in some matrix language like *MATLAB* [Moler 1981] or its derivatives *MatrixX* [Walker et al. 1982] and *CTRL-C* [Little et al. 1984]. A system is represented as a matrix quadruplet in *CTRL-C*. In *MatrixX* it is represented as a system matrix and an integer giving the system's order. It is, however, clear that it is not sufficient to only have matrices. A detailed discussion of this is given by Astroem (1984). There are a few additional data types, such as polynomials and transfer functions in *Blaise* [Delebecque & Steer 1985], *Impact* [Rimvall & Cellier (1984)] and *Eagles* [Gavel et al. 1986]. A more sophisticated data structure for systems was used in the Lund packages [Astroem 1985]. Our experiences indicate that it would be very useful to have even more flexible concepts.

## Simnon

The system description used in the simulation language Simnon [Elmqvist 1977], includes system descriptions for continuous and discrete time systems. A continuous system corresponds to state models described by an ordinary differential equation like

$$(1) \quad \frac{dx}{dt} = f(x, u, t)$$

$$y = g(x, u, t)$$

where  $x$  is the state vector,  $u$  the input vector and  $y$  the output vector. The corresponding *Simnon* representation is

```
CONTINUOUS SYSTEM <system identifier>
  INPUT <list of inputs>
  OUTPUT <list of outputs>
  STATE <list of states>
  DER <list of derivatives>
  TIME <variable>
  Computation of outputs
  Computation of derivatives
  Parameter assignment
  Initial value assignment
END
```

The standard state space model for a discrete time system is

$$(2) \quad x(t_{k+1}) = f(x_k, u_k, t_k)$$

$$y(t_k) = g(x_k, u_k, t_k)$$

where  $\{t_k\}$  is a sequence of sampling points. In Simnon such a system is described as

```
DISCRETE SYSTEM <system identifier>
  INPUT <list of inputs>
  OUTPUT <list of outputs>
  STATE <list of states>
  NEW <list of new states>
  TIME <variable>
  TSAMP <variable>
  Computation of outputs
  Computation of new state values
  Update the TSAMP-variable
  Modify states in continuous subsystems
  Parameter assignment
```

Initial value assignment  
**END**

Notice that this description is analogous to continuous systems. There is, however, a new variable (TSAMP ...) which gives the next time that the system should be sampled. In Simmon it is also possible to connect subsystems by using a connecting system, which is described by

**CONNECTING SYSTEM** <system identifier>  
**TIME** <variable>  
 Computation of inputs  
 Parameter assignment  
**END**

The Simmon notation is very natural for a control engineer. Long experience of using it has also shown that it is very easy to teach and use.

### **Discussion**

The matrix based languages lack a proper system concept. This means that it is difficult to implement operations which are naturally viewed as operations on a system. We cannot make the natural abstractions used in system theory and low level matrix operations have to be used instead. Simmon has a notion of systems. However, a drawback with its perspective is that it admits flat interconnections only. Particularly when dealing with large systems it would be desirable to allow for hierarchical interconnections. One possible approach has been suggested in Astroem (1985), and a more flexible solution is proposed in this paper.

### **3. Requirements**

We will now briefly discuss some key issues in system representation. An important requirement is that the descriptions we introduce should admit hierarchical system representations. Another is that it should be convenient to express systems composed of regular patterns of similar components in a convenient way..

To discuss suitable system descriptions we must also know how they are typically used. Typical operations on a system may seek to:

- Combine several subsystems into a new subsystem.
- Expand a system into its subsystems.
- Find interconnected loops.
- Compute steady state operating points.
- Compute steady-state input-output operations.
- Simulate.
- Linearize.
- Describe region of validity of a linearized model.
- Analyze stability, reachability and observability.
- Make a Kalman decomposition.

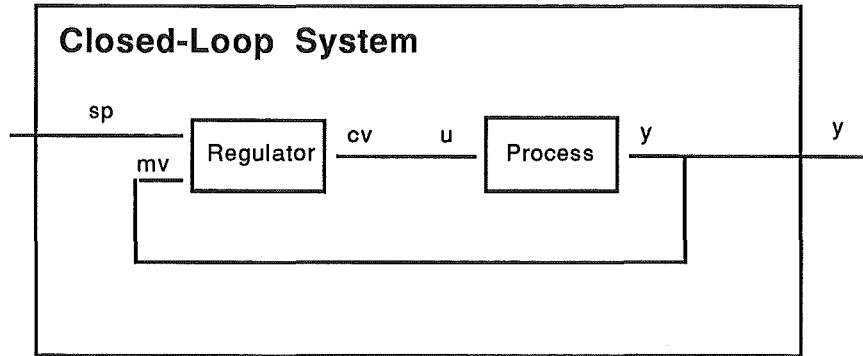
- Compute system inverses.
- Compute sensitivity functions.
- Compute well-conditioned linear representations.
- Find linear characteristics, such as
  - poles and zeros,
  - transfer functions,
  - frequency curves.
- Transform between alternative system representations.
- Perform and validate control design.
- Make model reductions.
- Fit parameters to experimental data.
- Find graphical representations.

Some of these operations are conveniently done numerically. Others require formula manipulation. It is therefore essential that the system can support numerical as well as formal calculations.

To describe systems it is thus necessary to have a rich structure which makes it possible to describe hierarchical interconnections of subsystems, where each subsystem is in turn composed of subsystems of its own. The subsystems may be of different types. They may be described in terms of state models, or as input-output relations like impulse responses or transfer functions. We also have a need for descriptions of different complexity.

#### 4. System Structure

Representation of system structure is a key element when dealing with complex systems. Graphical representations, like block diagrams, signal flow diagrams and bond graphs are common for this purpose. They can be used to present details of subsystems as well as to give an overview. In a block diagram description, as exemplified by figure 1, a subsystem is represented by a box and interconnections by lines between boxes. A line can denote a simple connection which tells us that the variables at the connection are the same. It can also represent a more complex situation, where several variables are involved. It is common practice to introduce arrows to indicate causality where this is possible. There may also be special symbols to denote simple operations such as addition and multiplication of signals. Many related descriptions like signal flow graphs and bond graphs abound in the literature.



**Figure 1:** Example of a hierarchical block diagram.  
This closed-loop system consists of 2  
subsystems: *Process* and *Regulator*.

To capture the essence of descriptions like block diagrams it becomes necessary to introduce the notions of subsystems and interconnections. In this paper we will only consider systems with well defined inputs and outputs. Such a system can be regarded as an abstraction of a box with input and output terminals. It can be represented as an object with the state variables

- name
- inputs
- outputs
- subsystems

The system in figure 1 can therefore be represented as follows.

```

name ClosedLoopSystem
  inputs r
  outputs y
  subsystems Regulator, Process

```

Inputs and outputs may be simple variables, but they could also be objects themselves, with properties such as units, range, ... . A primitive connection is a pair of input and output terminals, a concept which implies that the corresponding terminal variables are identical. To avoid ambiguity the name of the associated system is also attached to a connection, so that the notation (Regulator sp) denotes the *sp* terminal of the regulator component. The regulator as a unit has the representation

```

name Regulator

```

```

inputs mv, sp
outputs cv
subsystems nil

```

It has thus two inputs: *mv* (the measured value) and *sp* (the set point). It has one output: the controlled variable *cv*. The regulator has no subsystems. The process of figure 1 has the representation

```

name Process
inputs u
outputs y
subsystems nil

```

This notation is simple, natural and quite powerful.

## Methods

A system structure has a number of associated operations. Examples of basic low level constructor and mutator functions are

*MakeSystem, AddInputs, DeleteInputs, ...*

Basic query and selector functions of the type

*Inputs?, Inputs, ...*

are also needed to work with a system structure. The function *Inputs?* returns true if the subsystem has inputs and *Inputs* will return all inputs to a given subsystem. There are also primitive display functions like

*Show*

and a system editor which admits structured editing of system descriptions.

These functions operate on a single level only. For systems with subsystems it is also of interest to find all attributes of the system and all its associated subsystems. This is achieved through the functions

*AllInputs, AllOutputs, AllSubsystems, AllConnections*

Sometimes it is desirable to show these attributes in correspondence to the subsystem hierarchy.

*HierarchyOfInputs, HierarchyOfOutputs,  
HierarchyOfSubsystems, HierarchyOfConnections*



serve that purpose. Several functions explore system structure.

*InputsConnectedTo* and *OutputsConnectedTo*

return the systems connected to the input and output terminals of a given system.

*ContainedIn*

returns all systems in which a given subsystem is a component. A "loop" is a closed path which may be obtained by scanning a connection of subsystems in the direction defined by input-output causality. The functions

*Loop?*, *Loop*, *AllLoops*

may be used to tell whether a subsystem is part of a loop, to return such a loop itself, or all the possible loops associated with a given system.

## 5. System Behaviour

Only topological properties of a system, i.e. structure and interconnections, have been discussed so far. To adequately describe dynamic systems it is also necessary to specify how they behave. System behaviour is a very rich field. Examples of some relevant categories of behaviour are

- Static
- Qualitative
- State Space
- Stochastic State Space
- Stochastic Input Output
- Linear State Space
- Transfer Function
- Transient Response
- Describing Function

All of these categories can be described as objects. The static behaviour, for example, may be defined by a nonlinear function, and several methods are needed to work with static behaviour:

*FindOutput, FindInput, Linearize, MaxGain, MinGain,  
DegreeOfLinearity, OperatingRange, ...*

are some example of relevant functions. Qualitative behaviour attempts to describe some gross properties of a system, such as gain, time constants, and estimates of largest dynamic gain, some measure of nonlinearity, and some measure of how deterministic a system may be. We also need methods to obtain these properties from the more detailed representations. The ideas developed for automatic tuning of regulators are quite useful for this purpose [see Astroem & Haeggglund (1984)]. It is important to also allow qualitative descriptions, because it permits qualitative reasoning about system properties. In large systems with many subsystems, for example, we may choose to neglect an interconnection if a system with a very low gain is connected in parallel with a system with a very high gain. In a simulation which explores phenomena at a time scale of minutes we may be justified to use static models for subsystems whose time constants are less than one tenth of a second. It would also be useful to attach a *ValidityRange* property to a model. This could be described as a subset of the product of input and state spaces. With such a feature it becomes possible to write simulation programs which raise an exception condition if the state of a system moves outside its region of validity.

Many other types of behavioural analyses are possible and relevant. The state space approach and transfer function behaviour are well described in standard texts on control engineering. In this paper we will only explore some simple form of nonlinear state space behaviour.

nonlinear state space behaviour.

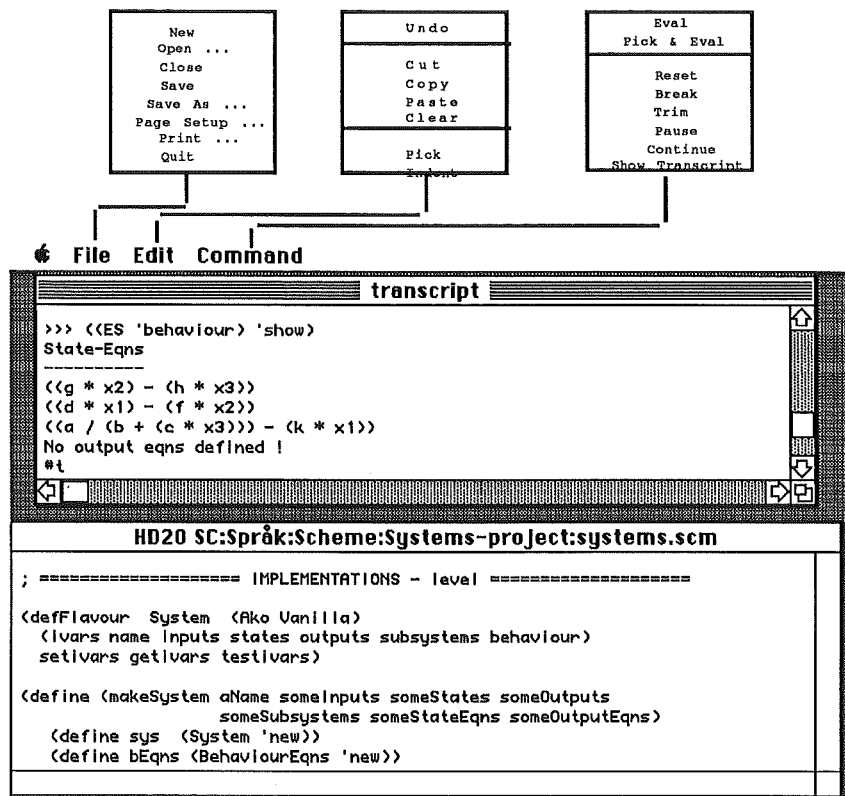
## **6. A Prototype Implementation**

A small prototype was used to test some of the ideas described in the previous sections. This program was written in Scheme on an Apple Macintosh computer, augmented by a so called "flavour system" to support an object-oriented style of program development. Our main goal was to experiment with descriptions of system structure, with some formula manipulation as a secondary objective.

At this stage our prototype handles only simple continuous time state behaviour, and little effort has gone into streamlining its user interface. We assume that all interactions occur within a Scheme workspace. Eventually we intend to provide a window-based, mouse and menu driven environment built around a simple model browser. Such an implementation is currently underway and uses the MacScheme+Toolsmith programming system [Semantic Microsystems (1986)].

### **The Scheme Programming Language.**

Scheme [Abelson et. al. (1985)] is a small and compact, as well as an extremely expressive and elegant programming language derived from Lisp. Its elegance stems from a small core of simple, orthogonal and very flexible concepts. In contrast to other Lisp systems Scheme offers lexical scoping, mandatory declaration of all objects, closures with persistent state, and continuations. Useful extensions, like object orientation, message passing, property inheritance and coroutines, can quickly and easily be grafted onto the language kernel. Like all dialects of Lisp, Scheme is an interactive language, fostering an exploratory style of program development. Structure editors and sophisticated debugging tools are normally provided as part of the programming environments in which it is embedded. Figure 2 shows a typical example taken from a session with MacScheme.

Figure 2: A session with *MacScheme*

Two windows are visible. The bottom one views a file named `systems.scm`, defining part of our systems toolbox (see appendix 2). Object definitions can be edited and evaluated here, using the pull-down menus elaborated in the figure. The top window shows a workspace called a transcript. It is used to interact directly with the Scheme interpreter. Expressions can be typed and evaluated there (i.e. during testing) and the system's responses are shown. In the figure the expression `((ES 'behaviour) 'show)` has been evaluated, returning a list of the system ES' behaviour equations. Editors incorporating some "knowledge" about the structure of valid Scheme expressions are usually part of such an environment. These tools can go a long way to remove the drawbacks of Lisp's sparse syntax; lightening the tasks of program design, implementation and debugging in spite of the proverbial jungle of parentheses.

## **Flavour Systems and Object Oriented Programming.**

Well designed programs should be built in layers. Such a methodology draws on sound principles of engineering design and helps to constrain the complexity of any large software system. After identifying a relevant layer, all primitive objects and operations at that level should be defined, possibly drawing on concepts bound at lower levels. These primitives may be provided through data structures and functions to create objects, select components, modify representations, evaluate predicates and display state information.

By encapsulating and localizing information as tightly as possible, object orientation [see, for example, Cox (1986)] carries this idea one step further. It seems a particularly appropriate metaphor for modelling applications, with many advantages over traditional methodologies. Most importantly, it changes our perception of programming towards viewing it as simulation of micro worlds, populated by autonomous objects which interact through message passing. Compared with traditional methodologies this new perspective on the programming process leads to a much closer correspondence between system specifications and implementations.

The key concepts of object oriented programming are the notions of closure, message passing, classification and inheritance. Its strongly modular object encapsulation encourages more transparent, more reliable, and more easily modifyable programs, whereas the penalties for object oriented system development are mainly rooted in their resource requirements.

The idea of structuring data into classes of objects was originally pioneered by the Simula programming language [Birtwistle et. al. (1973)] and has more recently been popularized through Smalltalk [Goldberg & Robson (1983)]. An object oriented program can be compared to a script for a play. It comprises a definition of all actors (classes of objects), as well as descriptions of their properties and behaviour in response to external stimuli (messages). The play (computation) itself may then be conceived as a sequence of interactions between the cast. In an object oriented program each relevant concept is therefore described in terms of three aspects:

- its (local) state, defined through its instance variables,
- the stimuli it will respond to, defined by methods,
- the kinds of objects it inherits from, defined by its superclasses.

All instance variables are completely encapsulated, so that they can only be accessed or modified through methods local to the class in which they are defined. While some languages, like Simula or C++, require that a class definition is recompiled each time a method is added, modified or deleted, so called flavour systems permit such changes to occur dynamically. Flavour systems are available as extensions to many dialects of Lisp. In order to support an object oriented programming style for our prototype, a flavour system for Scheme was designed and implemented at the University of Canterbury [Kreutzer & Stairmand 1989]. The following examples briefly illustrate its use. Appendix 1 shows a more detailed list of relevant features.

Suppose that we would like to implement a concept called "System", encapsulating information about its input, output and state, each represented by a list of symbols. We would like to be able to test whether instances of such a kind of system have a non-empty state, input or output; to return their current state, input or output; provide a new state, input or output; and print it in a more pleasing form. The following specification in Scheme+Flavours will provide these features:

```
(defFlavour System (ako Vanilla ) (ivars state input output)
  setivars getivars testivars)
```

This expression defines the System flavour, with instance variables called state, input and output.. Use of the optional keywords setivars, getivars, testivars then instructs the flavour system to create access methods for these instance variables. This causes the creation of methods state!, input!, output!, state; state, input, output; and state?, input?, output?. Vanilla is predefined by the flavour system and serves as the top of our inheritance network.

We can now explicitly defined additional methods; i.e. *show*.

```
(defMethod (show System ) ()
  (display "Components of this system are: ")
  (newline)
  (display "States   : ") (self 'state)
  (display "Inputs   : ") (self 'input)
  (display "Outputs  : ") (self 'output)
  nil )
```

Please note that evaluating these expressions does not create any systems yet. It rather defines how a particular flavour of system should look like. We may now proceed to create and interact with a number of system instances, i.e.

```

(makeInstance (Model System)
  ( (state '(x1 x2)) (input '(c))      (output '(y)) ) )
(makeInstance (Regulator System)
  ( (state '(i d))      (input '(r y)) (output '(u)) ) )

```

creates a Model and a Regulator object, with appropriately initialized instance variables. We can now send messages to these objects

```

(Model 'state)
; (x1 x2)
(Regulator 'input?)
; #t
(Regulator 'show)
; Components of this system are:
; States: (i d)
; Inputs: (r y)
; Outputs: (u)
; ()

(Model 'state! nil)
; ()
(Model 'state?)
; #f

```

Please note the Scheme convention under which methods with a "!" suffix change values ("side-effects"), whereas a "?" indicates predicates returning #t (for true) or #f (for false). Methods without suffix are used to non-destructively access and return information.

To demonstrate inheritance, let us now define *ActiveSystem* as a subflavour of *System*. Additional to all aspects of a generic system it will require a list of equations describing its behaviour.

```

(defFlavour ActiveSystem (ako System) (ivars behaviour)
  getivars testivars)

```

A flavour can have one or more superflavours, with *Vanilla* acting as a default. Inheritance is recursive, so that *ActiveSystem* inherits from both *System* and *Vanilla*. Any newly defined *ActiveSystem* will inherit all of a *System*'s instance variables and methods. For the *behaviour* instance variable we now want the system to create only access and predicate methods automatically. It should not be possible to change an *ActiveSystem*'s behaviour directly. Instead of the standard behaviour (return #t or #f) we also want to respond with "No information !" whenever the *behaviour?* message is received. This may be accomplished by "shadowing" the original method through a new definition.

```
(defMethod (behaviour? ActiveSystem ) ()
  (display "No information !") (newline))
```

We still want to be able to print such a flavour, but now this should include the value of its behaviour list. We will therefore redefine the `show` method. Because we are lazy, however, we want to make use of our previously defined `show` method for "ordinary" systems and just add the part for printing the behaviour equations. `self` is a so called "pseudo variable", bound to the currently executing instance of the flavour containing it. It can therefore be used by objects to pass messages to themselves. Unfortunately we can not send `'show` directly. Using it in that way within the body of our new `show` would create an infinite recursion, since we have "shadowed" the inherited "original" meaning. It can, however, be retrieved by sending `show` to our superflavours.

```
(defMethod (show ActiveSystem ) ()
  (self 'sendToSuperFlavours 'show nil)
  (display "This system's behaviour equations are:")
  (newline)
  (for-each (lambda (anEquation)
              (display anEquation) (newline))
    (self 'behaviour))
  nil )
```

We can test this scenario again by making a flavour and sending some messages:

```
(makeInstance (Model ActiveSystem)
  ((behaviour '( ((w * x2) (((- w * x1))
                      - (2 * (w * x2))) + (w * c)) )) )

(Model 'output)
; (y)
(Model 'behaviour?)
; No information !
(Model 'behaviour! '(x))
; Sorry, I don't know how to "behaviour!"
(Model 'behaviour)
; (((w * x2) (((- w * x1)) - (2 * (w * x2))) + (w * c)))
(Model 'show)
; Components of this system are:
; States: (x1 x2)
; Inputs: (c)
; Outputs: (y)
; This model's behaviour equations are:
; ((w * x2) (((- w * x1)) - (2 * (w * x2))) + (w * c))
; ()
```



## **An Example**

Our Prototype for system analysis is organized around a number of classes of objects. Figure 3 summarizes the messages applicable to each of them, together with relevant links defining a system's flavour/subflavour inheritance (bold lines) and "part of" hierarchies (dashed lines).

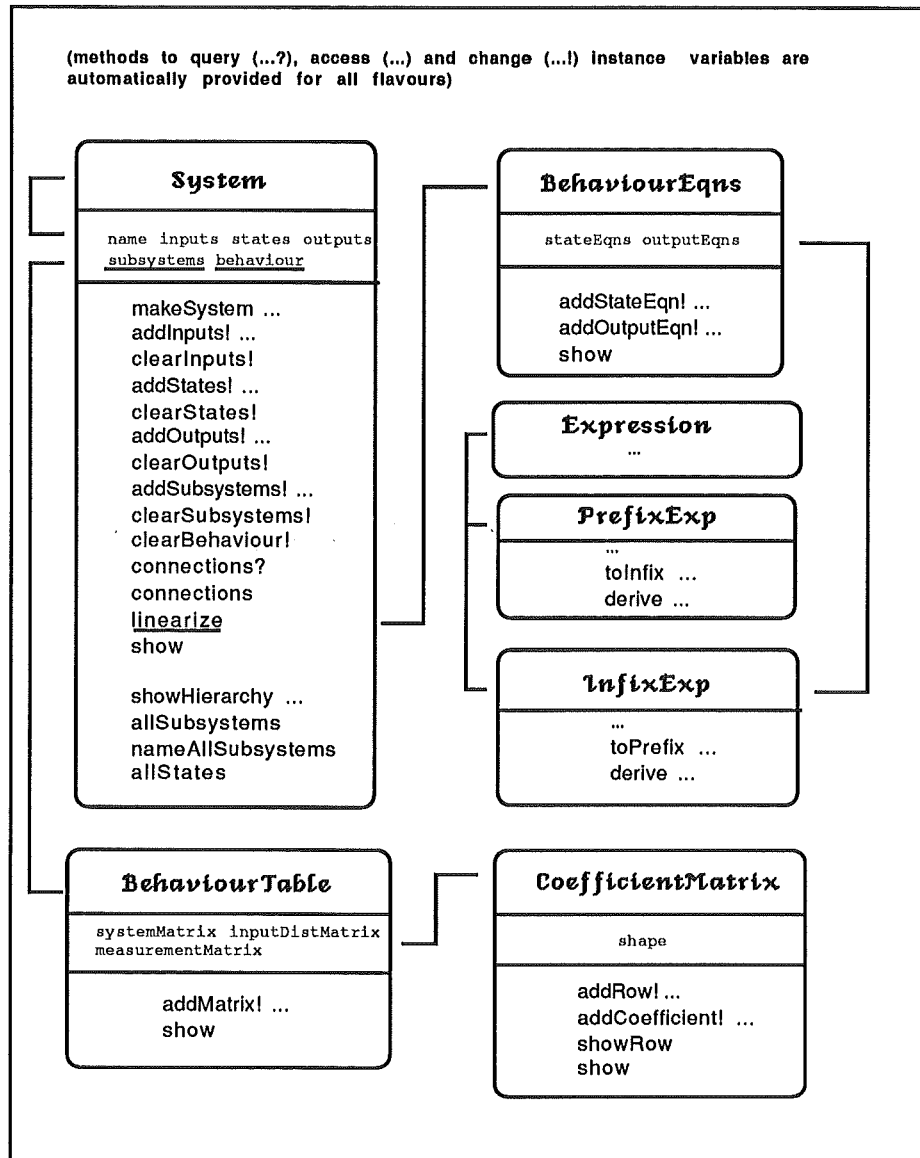


Figure 3: Objects & Protocol for a system analysis prototype

The figure shows that prefix and infix expressions are implemented as specializations of flavour expression and that no instance variables are defined for any of these. Expression flavours therefore serve only to encapsulate relevant operations. The expressions themselves are stored in lists. This was done to avoid the overhead of instantiating a great number of expressions produced "on the fly" by

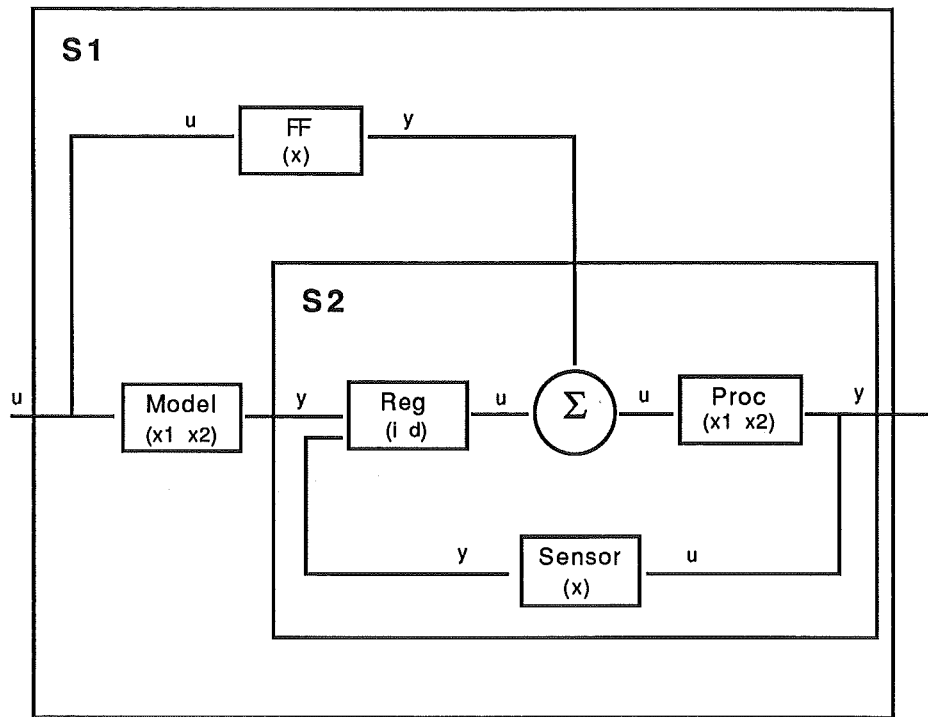
various methods in the system. The figure also shows that a system's subsystems are stored as a list of system objects and that its behaviour is encapsulated by an object of flavour BehaviourEqns. Behaviour equations contain expressions for state and output equations. The system method linearize returns a BehaviourTable, with its 3 matrices as instances of flavour CoefficientMatrix.

Each system contains six named components (slots):

*<name, inputs, states, outputs, subsystems, behaviour>.*

The prototype contains simple constructor, query, selector and display methods to manipulate such systems' properties. "Higher level" functions include methods to produce all subsystems for specific system instances. Operations to explore system structure and tools for linearization of system behaviour are also provided. A system's behaviour is characterized in terms of functions  $f$  and  $g$  in equation (1). The function  $f$  defines the rate of change of the system's state and  $g$  is its output map. Currently both have to be encoded as fully parenthesized expressions.

Consider a model of the control system shown in the block diagram depicted by figure 4. The system S1 contains subsystems called Model, FF and S2. S2 has three subsystems of its own: Reg, Proc and Sensor. Their inputs, outputs and subsystems are shown in the figure.



**Figure 4:** Example of a simple control system

State variables are introduced and the system's behaviour is described by functions  $f$  and  $g$ . Relevant properties of all bottom-level subsystems are summarized below:

- *Model (desired closed behaviour)*

**Input**  $u$

**Output**  $c$

**States**  $x_1, x_2$

**Behaviour:**

$$\frac{dx_1}{dt} = \omega x_1$$

$$\frac{dx_2}{dt} = \omega(-x_1 - 2\zeta x_2 + c)$$

$$y = x_1$$

- *FF (feedforward compensation)*

**Input**  $u$

**Output**  $y$

**State**  $x$

**Behaviour:**

$$\frac{dx}{dt} = -ax + (b-a)x$$

$$y = k(x+u)$$

- *Reg (simple PID regulator)*

**Input**  $r, y$

**Output**  $u$

**States**  $i, d$

**Behaviour:**

$$\frac{dd}{dt} = \frac{N}{T_d}(y-d)$$

$$\frac{di}{dt} = \frac{1}{T_i}(r-y)$$

$$u = k(r-y+i+N(d-y))$$

- *Process (process dynamics)*

**Input**  $u$

**Output**  $y$

**States**  $x_1, x_2$

**Behaviour:**

$$\frac{dx_1}{dt} = -a\sqrt{x_1} + bu$$

$$\frac{dx_2}{dt} = -a\sqrt{x_1} + c\sqrt{x_2}$$

$$y = x_2$$

• *Sensor (Sensor dynamics)*

**Input**     $u$   
**Output**  $y$   
**State**     $x$   
**Behaviour:**

$$\frac{dx}{dt} = \frac{1}{T}(u-x)$$

$$y = \frac{x}{1+x^2}$$

It was already mentioned that we have not yet put much effort into the user interface for this prototype. We are simply evaluating ordinary Scheme expressions. Our program will therefore need to specify this system in a somewhat tedious way; i.e. looking like this:

```

(define Model
  (makeSystem 'model '(c) '(x1 x2) '(y) nil
    '(((w * x2) (((- w * x1))
      - (2 * (w * x2))) + (w * c))
      '(x2) ))

(define FF
  (makeSystem 'ff '(u) '(x) '(y) nil
    '((( (- (B * x)) + ((a - b) * u)))
      '(((k * (x + u))) ))

(define Reg
  (makeSystem 'reg '(r y) '(i d) '(u) nil
    '((( (1 / Ti) * (r - y)) ((N / Td)
      * (d + y)))
      '(((K * (((r - y) + i) + (N * (d - y)))))))

(define Proc
  (makeSystem 'proc '(u) '(x1 x2) '(y) nil
    '((( (- (a * (x1 ** (1 / 2)))) + (b * u))
      ((a * (x1 ** (1 / 2)))
      - (a * (x2 ** (1 / 2)))) )
      '(((x2)) ))

```

```

(define Sensor
  (makeSystem 'sensor '(u) '(x) '(y) nil
              '(((u - x) / T))
              '((x / (1 + (x * x))))))

(define S2
  (makeSystem 's2 '(r u) nil '(y) (list reg proc sensor)
              nil nil))

(define S1
  (makeSystem 's1 '(c) nil '(y) (list s2 model ff) nil nil))

```

## A Sample Dialog

The simple dialogue below gives a flavour of how the prototype can be used. Assuming that the example model in figure 4 has been properly instantiated and initialized (see appendix 2), the expression (**S1 'show**) would create the following summary description:

```

=====
SYSTEM: s1
=====
Inputs   : (c)
Outputs  : (y)
States   : ()
*** Subsystems ***
+++++++
s2
model
ff
*** Connections ***
+++++++
(c y ())
*** Behaviour ***
+++++++
No state eqns defined !
No output eqns defined !
# t

```

There are several tools to explore a system's structure.

(**S2 'allSubsystems**) returns a list of all S2's subsystems, i.e. (Reg Proc Sensor). If this message is sent to a system whose components have subsystems of their own, then all are included.

(S1 'allSubsystems) therefore returns (S2 Model FF Reg Proc Sensor). Details about an object's component structure can be obtained by sending the message "showHierarchy". For example,

(S1 'showHierarchy 3) would provide:

```
+++++
SUBSYSTEMS of: s1
+++++
s2
  reg
  proc
  sensor
  model
  ff
  ( )
```

(Proc 'behaviour) returns a BehaviourEqns object. This can be printed in a readable fashion by ((Proc 'behaviour) 'show), which generates:

```
State-Eqns
-----
((a * (x1 ** (1 / 2))) - (a * (x2 ** (1 / 2))))
((a * (x1 ** (1 / 2))) + (b * u))
Output-Eqns
-----
(x 2)
( )
```

The message sequence ((Proc 'behaviour) 'stateEqns) returns function f, which defines the rate of state transformations, i.e.

```
((a * (x1 ** (1 / 2))) - (a * (x2 ** (1 / 2))))
((a * (x1 ** (1 / 2))) + (b * u))
```

The output map, g, of Proc can be obtained by  
((Proc ' behaviour) 'outputEqns), which will return

```
((x 2)).
```



Finally, a system can be linearized by sending the message "linearize", which generates a list of linearized equations stored in an object of class BehaviourTable. This can then be displayed as a set of 3 CoefficientMatrices, containing expressions.

((Proc 'linearize) 'show) therefore returns

```

*
..
..
*
.
.
*
..
-----      System Matrix      -----
(a * ((1 / 2) * (x1 ** ((1 / 2) - 1))))
0
(a * ((1 / 2) * (x1 ** ((1 / 2) - 1))))
(- (a * ((1 / 2) * (x2 ** ((1 / 2) - 1))))

-----      Input-Distribution Matrix      -----
b
0

-----      Measurement Matrix      -----
() ()

```

Please note the dots which are printed by this method. Each of these indicates the completion of a differentiation to compute one of the coefficients. This scheme offers the user some limited feedback on the method's progress during its time consuming execution.

## 8. Conclusions

The main purpose of this paper has been to explore new ways to describe interconnected systems, using object oriented programming as a structuring metaphor. It is relatively easy to implement and experiment with systems like the above prototype in Lisp. This ease of implementation is greatly aided by a powerful exploratory programming environment and an inheritance mechanism such as the one offered by the Canterbury Flavours system.

Our experiences so far vindicate our expectation that the proposed system descriptions are natural and easy to work with. A more complete system for system analysis can be implemented along the lines suggested in this paper. Some additional functionality will be needed and the extension the bidirectional interaction via keyboard and mouse events is relatively straightforward. The user interface will have to be improved substantially. Browsers, inspectors and debuggers must be provided and graphical representations of the type described by Elmqvist and Mattson (1986) should also be supported. Work along these lines is under way.

## References

- Abelson, H., Sussman, J. (with J. Sussman) (1985). *Structure and Interpretation of Computer Programs*. Cambridge (Mass.): MIT Press
- Allen, R. & Kreutzer, W. (1988). 'STRESS - A Smalltalk Rule Based Expert System Shell'. Proceedings *3rd New Zealand Conference on Expert Systems*. Wellington
- Astroem, K.J. (1983): "Computer Aided Modelling, Analysis and Design of Control Systems - A perspective," *IEEE Control Systems Magazine*, Vol. 3, May 1983, 4-16.
- Astroem, K.J. & Hagglund, T. (1984): "Automatic Tuning of Simple Regulators with Specifications on Phase and Amplitude Margins," *Automatica* 20, No. 5, 645-651.
- Astroem, K.J. (1985): "Computer Aided Tools for Control System Design - A perspective," in M. Jamshidi and C.J. Herget (Eds): *Computer-Aided Control Systems Engineering*, North Holland.
- Astroem, K.J. & Kreutzer, W. (1988). 'System Representation'. Report - *Department for Automatic Control, Lund Institute of Technology*. Lund (Sweden) 1988
- Bell, P.C. & O'Keefe, R.M. (1987). 'Visual Interactive Simulation - History, recent developments, and major issues'. *Simulation* Vol.49(3); 109-116
- Delebecque, F. & S. Steer (1985): "The Interactive System Blaise for Control Engineering," CADCE '85, *3rd IFAC/IFIP Symposium on Computer Aided Design in Control and Engineering Systems*, Lyngby, Denmark.
- Elmqvist, H. ((1975): "SIMNON - User's Manual," Report CODEN: LUTFD2/TFRT-3091, *Department of Automatic Control, Lund Institute of Technology*, Lund, Sweden.
- Elmqvist, H. (1978): "A Structured Model Language for Large Continuous Systems," Report CODEN: LUTFD2/TFRT-1015, *Department of Automatic Control, Lund Institute of Technology*, Lund, Sweden.
- Elmqvist, H. (1985): "LICS - Language for Implementation of Control Systems", Report CODEN: LUTFD2/TFRT-3179, *Department of Automatic Control, Lund Institute of Technology*, Lund, Sweden.
- Elmqvist, H. & Mattson, S.E. (1985): "A Simulator for Dynamical Systems Using Graphics and Equations for Modelling," *Proc. of the IEEE Control Systems Society Third Symposium on Computer-Aided Control System Design (CACSD)*, Arlington, Virginia.

Gavel, D.T., C.J. Herget, AND Lawyer, B.S. (1986): "The M Language - An Interactive Tool for Manipulating Matrices, Systems and Signals," *Dynamics and Controls Group, Engineering Research Division, Lawrence Livermore National Laboratory*, Livermore, California.

Goldberg, A. and Robson, D. (1983). *SMALLTALK-80 The Language and its Implementation*. Reading(Mass.): Addison Wesley

Integrated Systems Inc. (1984): "Matrix-X User's Guide, Matrix-X Reference Guide, Matrix-X Training Guide, Command Summary and On-line Help," *Integrated Systems, Inc.* 101 University Avenue, Palo Alto, California.

Kreutzer, W. (1986). *System Simulation. Programming Styles and Languages*. Reading(Mass.): Addison Wesley

Kreutzer (1988). 'A Modeller's Workbench - Simulation based on the desktop metaphor'. *Proceedings Artificial Intelligence and Simulation*. SanDiego. 43-48

Kreutzer, W. & McKenzie, B.J. (1989). *AI Programming. Tools & Metaphors*. Reading(Mass.): Addison Wesley

Lawyer, B. & Poggio, P. (1985): "EAGLES Requirements," *Computer Systems Research Group, Engineering Research Division, Lawrence Livermore National Laboratory*, Livermore, California.

Little, J.N., A. Emami-Noeini, and Bangert, S.N. (1984): "CTRL-C and Matrix Environments for the Computer-Aided Design of Control Systems," in Bensoussan and Lions (Eds.): *Analysis and Optimization of Systems*, Lecture Notes in Control and Information Sciences, Springer-Verlag, Berlin, Also in Jamshidi, M. and C.J. Herget (eds) *Computer-Aided Control Systems Engineering*, North-Holland, Amsterdam, The Netherlands, 111-124.

Minsky, M. (1985). 'Communication with Alien Intelligence'. *BYTE* (April 1985), 127-138

Moler, C.B. (1980): "MATLAB - User's Guide," *Department of Computer Science, University of New Mexico*, Albuquerque, U.S.A.

Rimvall, M. (1983): "IMPACT, Interactive Mathematical Program for Automatic Control Theory, User's Guide," *Department of Automatic Control, Swiss Federal Institute of Technology, ETH-Zentrum*, Zurich, Switzerland.

Rimvall, M. & Cellier, F. (1984): "IMPACT Interactive Mathematical Program for Automatic Control Theory," in Bensoussan and Lions (Eds.): *Analysis and Optimization of Systems*, Springer Lecture Notes in Control and Information Sciences, Springer, Berlin.

Rimvall, M. & Cellier, F. (1985): "A Structural Approach to CACSD," *Computer-Aided Control Systems Engineering*, North-Holland, Amsterdam, The Netherlands, pp. 149-158, In Jamshidi, M. and C.J. Herget (Eds).

Shah, S.C., Floyd, M.A. & Lehman, L.L. (1985): 'MATRIX-X: Control and Model Building CAE Capability,' in Jamshidi, M. and C.J. Herget (Eds.): *Computer-Aided Control Systems Engineering*, North-Holland, Amsterdam, The Netherlands, pp. 181-207.

Stairmand, M. & Kreutzer, W. (1988). 'POSE: A Process-Oriented Simulation Environment'. *Simulation* Vol.50(4); 143-153

Stefik, M. & Bobrow, D.G. (1985). Object-Oriented Programming: Themes and Variations. *The AI Magazine*. 40-62

Systems Control Technology (1984): "CTRL-C A Language for the Computer-Aided Design of Multivariable Control Systems, Users Guide," *Systems Control Technology*, 1801 Page Mill Road, Palo Alto, CA.

Vanbegin, M. & van Dooren, P. (1985): "MATLAB-SC, Appendix B: Numerical Subroutines for Systems and Control Problems," Technical Note N168, *Philips Research Laboratories*, Bosvoorde, Belgium.

Walker, R., Gregory, C. Jr. & Shah, S. (1984): "MATRIX-X: A Data Analysis, System Identification, Control Design and Simulation Package," *Integrated Systems, Inc.*, Palo Alto, California.

Walker, R., Shah, S. & Gupta, N.K. (1984): "Computer-Aided Engineering (CAE) for System Analysis," *Proc. of the IEEE*. 72, No. 12, 1732-1745.

## Appendix 1: *The "Canterbury Flavours" System - a Summary*

### A. Definition Macros

- new flavour definition (classes of objects)

```
(defFlavour aName (ako super-flavour ) (ivars aName ...)
  testivars getivars setivars)
```

{testivars, getivars, setivars are optional and can specified in any order. They control the automatic creation of test (...?), access (...) and change (...!) methods for instance variables}



**Appendix 2: Listing of Toolbox Functions**

```
; ===== IMPLEMENTATIONS - level =====
```

```
(defFlavour System (Ako Vanilla)
  (ivars name inputs states outputs subsystems behaviour)
  (setivars getivars testivars)

  (define (makeSystem aName someInputs someStates someOutputs
    someSubsystems someStateEqns someOutputEqns)
    (define sys (System 'new))
    (define bEqns (BehaviourEqns 'new))
    (for-each (lambda (x)
      (bEqns 'addStateEqn! x))
      someStateEqns)
    (for-each (lambda (x)
      (bEqns 'addOutputEqn! x))
      someOutputEqns)
    (sys 'name! aName)
    (sys 'inputs! someInputs)
    (sys 'states! someStates)
    (sys 'outputs! someOutputs)
    (sys 'subsystems! someSubsystems)
    (sys 'behaviour! bEqns)
    ; return the new system instance
    sys)

  (defMethod (addInputs! System) (anInputList)
    ; WARNING: Does not check for duplicates !
    (Set! inputs (APPEND (self 'inputs) anInputList)) )

  (defMethod (clearInputs! System) () (Set! inputs nil))

  (defMethod (addStates! System) (aStateList)
    ; WARNING: Does not check for duplicates !
    (Set! states (APPEND (self 'states) aStateList)) )

  (defMethod (clearStates! System) () (Set! states nil))

  (defMethod (addOutputs! System) (anOutputList)
    ; WARNING: Does not check for duplicates !
    (Set! outputs (APPEND (self 'outputs) anOutputList)) )

  (defMethod (clearOutputs! System) () (Set! outputs nil))
```

```

(defMethod (addSubsystems! System) (aSubsystemList)
; WARNING: Does not check for duplicates !
  (Set! subsystems (APPEND (self 'subsystems) aSubsystemList)) )

(defMethod (clearSubsystems! System) () (Set! subsystems nil))

(defMethod (clearBehaviour! System) () (Set! behaviour nil))

(defMethod (connections System) ()

  (define (removeDuplicatesFrom aList)
    (if (member (car aList) (cdr aList))
        (removeDuplicatesFrom (cdr aList))
        (begin (cons (car aList)
                      (if (null? aList)
                          nil
                          (removeDuplicatesFrom (cdr aList)))))))

  (RemoveDuplicatesFrom (Append (self 'inputs) (self 'outputs))) )

(defMethod (connections? System) ()
  (or (self 'inputs?) (self 'outputs?)))

(defMethod (show System) ()
  (display "=====") (newline)
  (display " SYSTEM: ")
  (display (self 'name)) (newline)
  (display "=====") (newline)
  (display " Inputs  : ") (display (self 'inputs)) (newline)
  (display " Outputs : ") (display (self 'outputs)) (newline)
  (display " States  : ") (display (self 'states)) (newline)
  (If (self 'subsystems?)
      (begin (display " *** Subsystems *** ") (newline)
              (display "      ++++++      ") (newline)
              (for-each (lambda (aSystem)
                          (if (not (null? aSystem))
                              (begin (display (aSystem 'name)) (newline))))
                    (self 'subsystems))) )
      (If (self 'connections?)
          (begin (display " *** Connections *** ") (newline)
                  (display "      ++++++      ") (newline)
                  (display (self 'connections)) (newline))) )
      (If (self 'behaviour?)
          (begin (display " *** Behaviour *** ") (newline)
                  (display (self 'behaviour)) (newline))) )

```

```

        (display "      ++++++++" ) (newline)
        ((self 'behaviour) 'show) (newline))) )

; ***** Implementation of "BEHAVIOUR-EQNS"

(defFlavour BehaviourEqns (ako Vanilla)
  (ivars stateEqns outputEqns) getivars setivars testivars)

(defMethod (addStateEqn! BehaviourEqns) (anExp)
  (Set! stateEqns (append (list anExp) (self 'stateEqns))))

(defMethod (addOutputEqn! BehaviourEqns) (anExp)
  (Set! outputEqns (append (list anExp) (self 'outputEqns))))

(defMethod (show BehaviourEqns) ()
  (If (self 'stateEqns?)
    (begin (display "State-Eqns") (newline)
            (display "-----") (newline)
            (for-each (lambda (x) (display x) (newline))
                      (self 'stateEqns)))
    (begin (display "No state eqns defined !") (newline)) )
  (If (self 'outputEqns?)
    (begin (display "Output-Eqns") (newline)
            (display "-----") (newline)
            (for-each (lambda (x) (display x) (newline))
                      (self 'outputEqns)))
    (begin (display "No output eqns defined !") (newline))) )

; ***** Implementation of (completely parenthesized)
"EXPRESSION"
; these flavours serve as "anchors" for relevant operations - the
; expressions themselves are kept as lists !!!

(defFlavour Expression (ako Vanilla))

(defMethod (prefix? Expression) (anExp)
  (if (member (car anExp) (list '+ '- '* '/' '**)) #t #f))
(defMethod (infix? Expression) (anExp)
  (if (member (cadr (anExp)) (list '+ '- '* '/' '**)) #t #f))

(defMethod (postfix? Expression) (anExp)
  (if (member (caddr (anExp)) (list '+ '- '* '/' '**)) #t #f))

(defMethod (constant? Expression) (anExp)
  (and (atom? anExp) (number? anExp)))

```



```

(defMethod (variable? Expression) (anExp)
  (and (atom? anExp) (symbol? anExp)))

(defMethod (expression? Expression) (anExp) (pair? anExp))

(defMethod (binary? Expression) (anExp)
  (if (self 'expression? anExp)
      (= (length anExp) 3)
      #f) )

(defMethod (unary? Expression) (anExp)
  (if (self 'expression? anExp)
      (= (length anExp) 2)
      #f) )

(defMethod (flat? Expression) (anExp)
  ; returns #t if an expression's "form" is not nested
  (define result #t)
  (cond ((atom? anExp) #t)
        ((null? anExp) #t)
        (else (set! result (atom? (CAR anExp)))
              (If result (self 'flat? (CDR anExp)))))) )

(defMethod (show Expression) (anExp)
  (display "the form of this ")
  (cond ((self 'unary? anExp) (display "unary "))
        ((self 'prefix? anExp) (display "prefix "))
        ((self 'infix? anExp) (display "infix "))
        ((self 'postfix? anExp) (display "postfix "))
        (else (display "IMPROPER ")))
  (display "expression is: ") (newline)
  (display anExp) (newline))

; ----- Unary Expressions -----

(defFlavour UnaryExp (ako Expression))

; ----- Binary Prefix Expressions ---

(defFlavour PrefixExp (ako Expression))

(defMethod (operator PrefixExp) (anExp)
  (if (self 'expression? anExp) (car anExp)))
(defMethod (firstArg PrefixExp) (anExp)

```



```

(string-append
  (symbol->string '-')
  (if (number? arg2)
      (number->string arg2)
      (symbol->string arg2))))
(list '- arg2)))
(list '- arg1 arg2)))
((Number? arg2) (If (Zero? arg2) arg1 (list '- arg1 arg2)))
(else (list '- arg1 arg2))) )

(define (difference? anExp)
  (and (self 'expression? anExp) (eq? (self 'operator anExp) '-)))

(define (makeProduct arg1 arg2)
  (Cond ((And (Number? arg1) (Number? arg2)) (* arg1 arg2))
        ((Number? arg1) (Cond ((Zero? arg1) 0)
                               ((=? 1 arg1) arg2)
                               ((=? -1 arg1) (list '- arg2))
                               (else (list '* arg1 arg2)))) )
        ((Number? arg2) (Cond ((Zero? arg2) 0)
                               ((=? 1 arg2) arg1)
                               ((=? -1 arg2) (list '- arg1))
                               (else (list '* arg1 arg2)))) )
        (else (list '* arg1 arg2))) )

(define (product? anExp)
  (and (self 'expression? anExp) (eq? (self 'operator anExp) '*)))

(define (makeQuotient arg1 arg2)
  (Cond ((And (Number? arg1) (Number? arg2)) (/ arg1 arg2))
        ((Number? arg1) (If (Zero? arg1) 0 (list '/ arg1 arg2)))
        ((Number? arg2)
         (Cond ((Zero? arg2)
                (display "Division by zero !!!") (newline) (reset))
               ((=? 1 arg2) arg1)
               (else (list '/ arg1 arg2))))
        (else (list '/ arg1 arg2))) )

(define (quotient? anExp)
  (and (self 'expression? anExp) (eq? (self 'operator anExp) '/)))

(define (makePower aBase anExp)
  (Cond ((And (Number? aBase) (Number? anExponent))
        (expt aBase anExp))
        ((Number? aBase) (Cond ((Zero? aBase) 0)

```

```

                ((=? 1 aBase) 1)
                (else (list '** aBase anExp))))
((Number? anExp) (Cond ((Zero? anExp) 1)
                        ((=? 1 anExp) aBase)
                        (else (list '** aBase anExp))))
(else (list '** aBase anExp)) ))

(define (power? anExp)
  (and (self 'expression? anExp) (eq? (self 'operator anExp) '**)))

(cond ; expression is a constant or variable
      ; - return 1 if same as "aVar", 0 otherwise
      ((self 'constant? anExp) 0)
      ((self 'variable? anExp) (If (SameVar? anExp aVar) 1 0))
      ; expression is not in prefix form - reject !
      ((not (dummy 'prefix? anExp))
       (display "NOT a prefix exp -- rejected") nil)
      ; expression is a sum or difference -
      ; add or subtract their differentials
      ((sum? anExp)
       (makeSum (self 'derive (self 'firstArg anExp) aVar)
                 (self 'derive (self 'secondArg anExp) aVar)) )
      ((difference? anExp)
       (makeDifference (self 'derive (self 'firstArg anExp) aVar)
                       (self 'derive (self 'secondArg anExp) aVar)) )
      ; expression is a Product - multiply partners' differentials
      ; and add
      ((product? anExp)
       (makeSum (makeProduct (self 'firstArg anExp)
                              (self 'derive (self 'secondArg anExp) aVar))
                 (makeProduct (self 'secondArg anExp)
                              (self 'derive (self 'firstArg anExp) aVar))) )
      ; expression is a Quotient - differentiate product of first and
      ; (1 over second arg)
      ((quotient? anExp)
       (self 'derive (makeProduct (self 'firstArg anExp)
                                   (makePower (self 'secondArg anExp) -1))
               aVar))
      ; expression is a Power - differentiate product of exponent
      ; and (one less than original exponentiation)
      ((power? anExp)
       (makeProduct (self 'secondArg anExp)
                     (makeProduct
                      (makePower (self 'firstArg anExp)
                                (makeDifference (self 'secondArg anExp) 1))

```

```

        (self 'derive (self 'firstArg anExp) aVar)))))) ))

; ----- Binary Infix Expressions -----

(defFlavour InfixExp (ako Expression))

(defMethod (operator InfixExp) (anExp)
  (if (self 'binary? anExp) (cadr anExp)))
(defMethod (firstArg InfixExp) (anExp)
  (if (self 'binary? anExp) (car anExp)))

(defMethod (secondArg InfixExp)
  (anExp) (if (self 'binary? anExp) (caddr anExp)))

(defMethod (toPrefix InfixExp) (anInfixExp)
  ;converts a prefix expression to an infix expression
  (Cond ((Null? anInfixExp) nil)
    ((Atom? anInfixExp) anInfixExp)
    ((self 'unary? anInfixExp) (list (self 'operator anInfixExp)
                                      (self 'toPrefix
                                           (self 'firstArg anInfixExp))))
    ((self 'flat? anInfixExp) (list (self 'operator anInfixExp)
                                      (self 'firstArg anInfixExp)
                                      (self 'secondArg anInfixExp)))
    (else (list (self 'operator anInfixExp)
                 (self 'toPrefix (self 'firstArg anInfixExp))
                 (self 'toPrefix (self 'secondArg anInfixExp)))))) ))

(defMethod (derive InfixExp) (anInfixExp aVar)
  (define dummy (PrefixExp 'new))
  (dummy 'toInfix (dummy 'derive (self 'toPrefix anInfixExp) aVar)) )

; ----- Binary Postfix Expressions -----

; NOT YET IMPLEMENTED

; ===== APPLICATIONS - Level =====

(defMethod (showHierarchy System) (aTabLevel)

  (define (showSubsystems aSystemList someSpaces aTabLevel)
    (define (printSpaces someSpaces)
      (do ((i 0 (1+ i))) ((=? i someSpaces) nil) (display " ")))

```

```

(if (null? aSystemList)
  nil
  (begin (printSpaces someSpaces)
    (display ((car aSystemList) 'name)) (newline)
    (showSubsystems ((car aSystemList) 'subSystems)
      (+ someSpaces aTabLevel)
      aTabLevel)
    (showSubsystems (cdr aSystemList)
      someSpaces
      aTabLevel)))) )

(display "+++++")
(newline)
(display "SUBSYSTEMS of: ") (display (self 'name)) (newline)
(display "+++++")
(newline)
(if (self 'subsystems?)
  (showSubsystems (self 'subSystems) 0 aTabLevel)
  nil) )

(defMethod (allSubsystems System) ()
; returns a list of all the system's subsystems

(define (traceAllSubsystems aSystemCollection)
; returns a list of all the subsystems of all systems in a collection
; (depth first search)

(define (removeDuplicatesFrom aList)
  (if (member (car aList) (cdr aList))
    (removeDuplicatesFrom (cdr aList))
    (begin (cons (car aList)
      (if (null? aList)
        nil
        (removeDuplicatesFrom (cdr aList)))))))

(if (not (null? aSystemCollection))
  (begin (removeDuplicatesFrom
    (append
      ; find the first system's subsystems
      ((eval (car aSystemCollection)) 'subsystems)
      ; now find the subsystems of those
      (traceAllSubsystems
        ((eval (car aSystemCollection)) 'subsystems))
      ; process the rest of the list

```

```

        (traceAllSubsystems (cdr aSystemCollection)) )))) )

(traceAllSubsystems (list (self 'name))) )

(defMethod (nameAllSubsystems System) ()
  (for-each (lambda (aSystem) (display (aSystem 'name)) (newline))
    (self 'allSubsystems)) )

(defMethod (allStates System) ()
  ; returns a list of all the system's states

  (define (traceAllStates aSystemCollection)
    ; returns a list of states for all systems in a collection
    ; AND all their subsystems (depth first search)

    (if (null? aSystemCollection)
      nil
      (begin (list
        ; find the first system's states
        ((eval (car aSystemCollection)) 'states)
        ; recurse down the subsystems and return their states
        (if (not (null? ((eval (car aSystemCollection))
          'subsystems)))
          (begin
            ((car
              ((eval (car aSystemCollection)) 'subsystems))
              'states)
            (traceAllStates
              (cdr ((eval (car aSystemCollection)) 'subsystems))))
          ; now recurse down the tails of the collection
          (traceAllStates (cdr aSystemCollection))))))

    (traceAllStates (list (self 'name))) )

(defMethod (linearize System) ()
  ; linearizes a system's behaviour by partial differentiation across all
  ; equations, state, input and output variables.
  ; It returns a BehaviourTable object.

  (define table (BehaviourTable 'new))

  (define (differentiateAcross someEqns someVars)

    (define dummy (InfixExp 'new))
    (define matrix (CoefficientMatrix 'new))

```

```

(define row '())

(display "**") (newline)
(for-each
  (lambda (nextExp)
    (for-each
      (lambda (nextVar)
        (set! row (matrix 'addCoefficient!
                          row
                          (dummy 'derive nextExp nextVar)))
        (display ".")
        someVars)
      (matrix 'addRow! row)
      (set! row nil)
      (newline))
    someEqns) matrix)

; process the systemMatrix
(table 'addMatrix! 'system
      (differentiateAcross ((self 'behaviour) 'stateEqns)
                          ( self 'states)))

; process the inputDistMatrix
(table 'addMatrix! 'inputDist
      (differentiateAcross ((self 'behaviour) 'stateEqns)
                          ( self 'inputs)))

; process the measurementMatrix
(table 'addMatrix! 'measurement
      (differentiateAcross ((self 'behaviour) 'outputEqns)
                          ( self 'states)))

table)

; ***** Implementation of "BEHAVIOUR-TABLE"

(defFlavour BehaviourTable (ako Vanilla)
  (ivars systemMatrix inputDistMatrix measurementMatrix)
  (setivars getivars testivars)
  ; systemMatrix      : n*n coefficients - across all n states
  ; inputDistMatrix   : n*m coefficients - across all n states & all
  ; m inputs
  ; measurementMatrix: k*n coefficients - across all k outputs & all
  ; n states

  (defMethod (addMatrix! BehaviourTable) (aName aMatrix)
    (cond ((eq? aName 'system)
           (self 'systemMatrix! aMatrix))

```



```

      ((eq? aName 'inputDist)
       (self 'inputDistMatrix! aMatrix))
      ((eq? aName 'measurement)
       (self 'measurementMatrix! aMatrix))) nil)

(defMethod (show BehaviourTable) ()
  (display "----- System Matrix -----") (newline)
  (if (self 'systemMatrix?)
      ((self 'systemMatrix) 'show)
      (display "+++ empty !"))
  (newline)
  (display "----- Input-Distribution Matrix -----") (newline)
  (if (self 'inputDistMatrix?)
      ((self 'inputDistMatrix) 'show)
      (display "+++ empty !"))
  (newline)
  (display "----- Measurement Matrix -----") (newline)
  (if (self 'measurementMatrix?)
      ((self 'measurementMatrix) 'show)
      (display "+++ empty !"))
  (newline) nil)

; ***** Implementation of "COEFFICIENTMATRIX"

(defFlavour CoefficientMatrix (ako Vanilla) (ivars shape)
  getivars setivars testivars)
; shape: 2 dimensional table of coefficients
; (lists of expressions or 0)
; note: expressions are not (!) represented as objects
; --- used for all 3 components of BehaviourTables ---

(defMethod (addRow! CoefficientMatrix) (aRowList)
  (self 'shape! (append (list aRowList) (self 'shape))))

(defMethod (addCoefficient! CoefficientMatrix)
  (aRow aCoefficient)
  (append aRow (list aCoefficient)))

(defMethod (showRow CoefficientMatrix) (aList)
  (If (null? aList)
      (display "--- no coefficients in row ---")
      (for-each (lambda (element) (display element) (display " "))
                 aList)) )

```

```
(defMethod (show CoefficientMatrix) ()  
  (for-each (lambda (row) (self 'showRow row) (newline))  
    (self 'shape)) )
```